

PATENT APPLICATION

HISTOGRAM PERFORMANCE COUNTERS FOR USE IN
TRANSACTION LATENCY ANALYSIS

Inventor: Adnan Khaleel of
Austin, Texas
Citizen of India

Assignee: Newisys, Inc.
A Delaware corporation

BEYER WEAVER & THOMAS, LLP
P.O. Box 778
Berkeley, California 94704-0778
(510) 843-6200

HISTOGRAM PERFORMANCE COUNTERS FOR USE IN TRANSACTION LATENCY ANALYSIS

BACKGROUND OF THE INVENTION

The present invention relates generally to measuring transaction latency in computer systems. More specifically, the present invention provides techniques for accurately monitoring transaction latency in a computer system on an ongoing basis with little or no disturbance of the system state.

A transaction in a computer system, e.g., a memory transaction or an I/O transaction, is characterized by a latency which corresponds to the time which elapses between the initiation and completion of the transaction. The accurate determination of such latencies is critical to measuring system performance, for evaluating the effects of optimizations and design refinements, and as a measure of the scalability of the system.

The way in which transaction latency has been measured conventionally is by connecting an external test apparatus, e.g., logic analyzer, to the computer system's memory bus, recording when a transaction begins (e.g., by detecting a read request), recording when the transaction is completed (e.g., by detecting data returned in response to the read request), and counting the number of clock cycles between the two events. Such an approach has several limitations.

First, correlating the first event with the second is a nontrivial problem in that there are typically billions and billions of transactions flowing through the system. Second, the instrumentation associated with even a single access point is cumbersome. That is, test leads from a typically bulky and non-portable piece of test equipment must be connected to as

many points in the system at which transactions are generated as possible. This severely limits the contexts in which this approach may be employed, particularly for systems involving multiple processors.

Another limitation with conventional approaches to measuring latency relates to the fact that the measuring system (typically user software) must periodically poll the hardware device under test in order to get an averaged value (the real metric of interest). Since making each measurement adds some overhead to the system performance that cannot be ignored, the user generally has to make a trade off between accuracy and perturbing the system state. This can lead to inaccurate results.

As discussed above, latency measurements are useful during the design and verification of a system. Such measurements could also be potentially useful during run-time operation. However, for at least the reasons discussed above, conventional approaches to measuring latency are simply not practical for providing such run-time measurements.

In view of the foregoing, techniques for measuring latency in computer systems which avoid some or all of the limitations described above are desirable.

SUMMARY OF THE INVENTION

According to the present invention, methods and apparatus are provided for measuring latency in computer systems. According to a specific embodiment, a computer system is provided which includes a processor and a memory. The processor is operable to initiate transactions involving the memory. The computer system further includes a latency counter operable to generate a latency count for each of selected ones of the transactions. The system also includes a plurality of histogram counters. Each histogram counter is operable to count selected ones of the latency counts corresponding to an associated latency

range. According to some embodiments, such latency data may also be generated for I/O transactions.

According to other embodiments, methods and apparatus are provided for generating latency information for any electronic system. According to one embodiment, an electronic system characterized by a plurality of transactions is provided. The electronic system includes a latency counter operable to generate a latency count for each of selected ones of the transactions. The system also includes a plurality of histogram counters. Each histogram counter is operable to count selected ones of the latency counts corresponding to an associated latency range.

A further understanding of the nature and advantages of the present invention may be realized by reference to the remaining portions of the specification and the drawings.

BRIEF DESCRIPTION OF THE DRAWINGS

Figs. 1A and 1B are diagrammatic representations depicting systems having multiple clusters.

Fig. 2 is a diagrammatic representation of an exemplary cluster having a plurality of processors.

Fig. 3 is a diagrammatic representation of an exemplary interconnection controller.

Fig. 4 is a diagrammatic representation of an exemplary local processor.

Fig. 5 is a diagrammatic representation of an exemplary memory mapping scheme.

Fig. 6A is a simplified block diagram of an exemplary four cluster system.

Fig. 6B is a combined routing table including routing information for the four cluster system of Fig. 6A.

Figs. 7 and 8 are flowcharts illustrating transaction management in a multi-cluster system.

Fig. 9 is a diagrammatic representation of communications relating to an exemplary transaction in a multi-cluster system.

Fig. 10 is another diagrammatic representation of an exemplary interconnection controller.

Fig. 11 is an exemplary mapping of protocol engines in a processor cluster to a global memory space in a multi-cluster system.

Fig. 12 is a flowchart illustrating mapping of transactions to protocol engines.

Fig. 13 is a block diagram of an interconnection controller implemented according to a specific embodiment of the invention.

Fig. 14 is a flowchart illustrating operation of a specific embodiment of the invention.

Fig. 15 is block diagram of an exemplary shared-bus computer system in which embodiments of the invention may be implemented.

DETAILED DESCRIPTION OF SPECIFIC EMBODIMENTS

Reference will now be made in detail to some specific embodiments of the invention including the best modes contemplated by the inventors for carrying out the invention.

Examples of these specific embodiments are illustrated in the accompanying drawings.

While the invention is described in conjunction with these specific embodiments, it will be

understood that it is not intended to limit the invention to the described embodiments. On the contrary, it is intended to cover alternatives, modifications, and equivalents as may be included within the spirit and scope of the invention as defined by the appended claims.

Multi-processor architectures having point-to-point communication among their processors are suitable for implementing specific embodiments of the present invention. In the

following description, numerous specific details are set forth in order to provide a thorough

understanding of the present invention. The present invention may be practiced without some or all of these specific details. Well known process operations have not been described in detail in order not to unnecessarily obscure the present invention. Furthermore, the present application's reference to a particular singular entity includes that possibility that the methods and apparatus of the present invention can be implemented using more than one entity, unless the context clearly dictates otherwise.

The present invention provides techniques by which latency data may be generated for any of a wide variety of electronic or data processing systems. Figs. 1-12 are diagrams and flowcharts describing specific computer architectures in which the techniques of the present invention may be implemented. It should be noted, however, that the invention is widely applicable and should not be limited to the architectures described.

Fig. 1A is a diagrammatic representation of one example of a multiple cluster, multiple processor system which may employ the techniques of the present invention. Each processing cluster 101, 103, 105, and 107 includes a plurality of processors. The processing clusters 101, 103, 105, and 107 are connected to each other through point-to-point links 111a-f. The multiple processors in the multiple cluster architecture shown in Fig. 1A share a global memory space. In this example, the point-to-point links 111a-f are internal system connections that are used in place of a traditional front-side bus to connect the multiple processors in the multiple clusters 101, 103, 105, and 107. The point-to-point links may support any point-to-point coherence protocol.

Fig. 1B is a diagrammatic representation of another example of a multiple cluster, multiple processor system that may employ the techniques of the present invention. Each processing cluster 121, 123, 125, and 127 is coupled to a switch 131 through point-to-point links 141a-d. It should be noted that using a switch and point-to-point links allows implementation with fewer point-to-point links when connecting multiple clusters in the

system. A switch 131 can include a general purpose processor with a coherence protocol interface. According to various implementations, a multi-cluster system shown in Fig. 1A may be expanded using a switch 131 as shown in Fig. 1B.

Fig. 2 is a diagrammatic representation of a multiple processor cluster such as, for example, cluster 101 shown in Fig. 1A. Cluster 200 includes processors 202a-202d, one or more Basic I/O systems (BIOS) 204, a memory subsystem comprising memory banks 206a-206d, point-to-point communication links 208a-208e, and a service processor 212. The point-to-point communication links are configured to allow interconnections between processors 202a-202d, I/O switch 210, and interconnection controller 230. The service processor 212 is configured to allow communications with processors 202a-202d, I/O switch 210, and interconnection controller 230 via a JTAG interface represented in Fig. 2 by links 214a-214f. It should be noted that other interfaces are supported. I/O switch 210 connects the rest of the system to I/O adapters 216 and 220, and to BIOS 204 for booting purposes.

According to specific implementations, the service processor has the intelligence to partition system resources according to a previously specified partitioning schema. The partitioning can be achieved through direct manipulation of routing tables associated with the system processors by the service processor which is made possible by the point-to-point communication infrastructure. The routing tables can also be changed by execution of the BIOS code in one or more processors. The routing tables are used to control and isolate various system resources, the connections between which are defined therein.

The processors 202a-d are also coupled to an interconnection controller 230 through point-to-point links 232a-d. According to various implementations and as will be described below in greater detail, interconnection controller 230 performs a variety of functions which enable the number of interconnected processors in the system to exceed the node ID space and mapping table limitations associated with each of a plurality of processor clusters.

According to some implementations, interconnection controller 230 performs a variety of other functions including the maintaining of cache coherency across clusters.

Interconnection controller 230 can be coupled to similar controllers associated with other multi-processor clusters. It should be noted that there can be more than one such

5 interconnection controller in one cluster. Interconnection controller 230 communicates with both processors 202a-d as well as remote clusters using a point-to-point protocol.

More generally, it should be understood that the specific architecture shown in Fig. 2 is merely exemplary and that implementations are contemplated having different configurations and resource interconnections, and a variety of alternatives for each of the system resources shown. However, for purpose of illustration, specific details of cluster 200 will be assumed. For example, most of the resources shown in Fig. 2 are assumed to reside on a single electronic assembly. In addition, memory banks 206a-206d may comprise double data rate (DDR) memory which is physically provided as dual in-line memory modules (DIMMs). I/O adapter 216 may be, for example, an ultra direct memory access 10 (UDMA) controller or a small computer system interface (SCSI) controller which provides access to a permanent storage device. I/O adapter 220 may be an Ethernet card adapted to provide communications with a network such as, for example, a local area network (LAN) or the Internet. BIOS 204 may be any persistent memory like flash memory.

According to one implementation, service processor 212 is a Motorola MPC855T 20 microprocessor which includes integrated chipset functions, and interconnection controller 230 is an Application Specific Integrated Circuit (ASIC) supporting the local point-to-point coherence protocol. Interconnection controller 230 can also be configured to handle a non-coherent protocol to allow communication with I/O devices. In one implementation, interconnection controller 230 is a specially configured programmable chip such as a 25 programmable logic device or a field programmable gate array. In another implementation,

the interconnect controller 230 is an Application Specific Integrated Circuit (ASIC). In yet another implementation, the interconnect controller 230 is a general purpose processor augmented with an ability to access and process interconnect packet traffic.

Fig. 3 is a diagrammatic representation of one example of an interconnection controller 230 which may facilitate various aspects of the present invention. According to various implementations, the interconnection controller includes a protocol engine 305 configured to handle packets such as probes and requests received from processors in various clusters of a multi-processor system. The functionality of the protocol engine 305 can be partitioned across several engines to improve performance. In one example, partitioning is done based on packet type (request, probe and response), direction (incoming and outgoing), or transaction flow (request flows, probe flows, etc).

The protocol engine 305 has access to a pending buffer 309 that allows the interconnection controller to track transactions such as recent requests and probes and associate the transactions with specific processors. Transaction information maintained in the pending buffer 309 can include transaction destination nodes, the addresses of requests for subsequent collision detection and protocol optimizations, response information, tags, and state information. As will become clear, this functionality is leveraged to enable particular aspects of the present invention.

The interconnection controller has a coherent protocol interface 307 that allows the interconnection controller to communicate with other processors in the cluster as well as external processor clusters. The interconnection controller may also include other interfaces such as a non-coherent protocol interface 311 for communicating with I/O devices (e.g., as represented in Fig. 2 by links 208c and 208d). According to various implementations, each interface 307 and 311 is implemented either as a full crossbar or as separate receive and transmit units using components such as multiplexers and buffers. It should be noted that

the interconnection controller 230 does not necessarily need to provide both coherent and non-coherent interfaces. It should also be noted that an interconnection controller 230 in one cluster can communicate with an interconnection controller 230 in another cluster.

According to various implementations, processors 202a-202d are substantially identical. Fig. 4 is a simplified block diagram of such a processor 202 which includes an interface 402 having a plurality of ports 404a-404c and routing tables 406a-406c associated therewith. Each port 404 allows communication with other resources, e.g., processors or I/O devices, in the computer system via associated links, e.g., links 208a-208e of Fig. 2.

The infrastructure shown in Fig. 4 can be generalized as a point-to-point, distributed routing mechanism which comprises a plurality of segments interconnecting the systems processors according to any of a variety of topologies, e.g., ring, mesh, etc. Each of the endpoints of each of the segments is associated with a connected processor which has a unique node ID and a plurality of associated resources which it “owns,” e.g., the memory and I/O to which it’s connected.

The routing tables associated with each of the nodes in the distributed routing mechanism collectively represent the current state of interconnection among the computer system resources. Each of the resources (e.g., a specific memory range or I/O device) owned by any given node (e.g., processor) is represented in the routing table(s) associated with the node as an address. When a request arrives at a node, the requested address is compared to a two level entry in the node’s routing table identifying the appropriate node and link, i.e., given a particular address within a range of addresses, go to node x; and for node x use link y.

As shown in Fig. 4, processor 202 can conduct point-to-point communication with three other processors according to the information in the associated routing tables.

According to a specific implementation, routing tables 406a-406c comprise two-level tables.

a first level associating the unique addresses of system resources (e.g., a memory bank) with a corresponding node (e.g., one of the processors), and a second level associating each node with the link (e.g., 208a-208e) to be used to reach the node from the current node.

Processor 202 also has a set of JTAG handshake registers 408 which, among other things, facilitate communication between the service processor (e.g., service processor 212 of Fig. 2) and processor 202. That is, the service processor can write routing table entries to handshake registers 408 for eventual storage in routing tables 406a-406c.

As mentioned above, the basic protocol upon which the clusters in specific implementations are based provides for a limited node ID space which, according to a particular implementation, is a 3-bit space, therefore allowing for the unique identification of only 8 nodes. That is, if this basic protocol is employed without the innovations described herein, only 8 nodes may be interconnected in a single cluster via the point-to-point infrastructure. To get around this limitation, a hierarchical mechanism is introduced which preserves the single-layer identification scheme within particular clusters while enabling interconnection with and communication between other similarly situated clusters and processing nodes.

According to a specific implementation, one of the nodes in each multi-processor cluster is an interconnection controller, e.g., interconnection controller 230 of Fig. 2, which manages the hierarchical mapping of information thereby enabling multiple clusters to share a single memory address space while simultaneously allowing the processors within its cluster to operate and to interact with any processor in any cluster without “knowledge” of anything outside of their own cluster. The interconnection controller appears to its associated processor to be just another one of the processors or nodes in the cluster.

In the basic protocol, when a particular processor in a cluster generates a request, a set of address mapping tables are employed to map the request to one of the other nodes in

the cluster. That is, each node in a cluster has a portion of a shared memory space with which it is associated. There are different types of address mapping tables for main memory, memory-mapped I/O, different types of I/O space, etc. These address mapping tables map the address identified in the request to a particular node in the cluster.

5 A set of routing tables are then employed to determine how to get from the requesting node to the node identified from the address mapping table. That is, as discussed above, each processor (i.e., cluster node) has associated routing tables which identify a particular link in the point-to-point infrastructure which may be used to transmit the request from the current node to the node identified from the address mapping tables. Although
10 generally a node may correspond to one or a plurality of resources (including, for example, a processor), it should be noted that the terms node and processor are often used interchangeably herein. According to a particular implementation, a node comprises multiple sub-units, e.g., CPUs, memory controllers, I/O bridges, etc., each of which has a unit ID.

15 In addition, because individual transactions may be segmented in non-consecutive packets, each packet includes a unique transaction tag to identify the transaction with which the packet is associated with reference to the node which initiated the transaction. According to a specific implementation, a transaction tag identifies the source node (3-bit field), the source node unit (2-bit field), and a transaction ID (5-bit field).

20 Thus, when a transaction is initiated at a particular node, the address mapping tables are employed to identify the destination node (and unit) which are then appended to the packet and used by the routing tables to identify the appropriate link(s) on which to route the packet. The source information is used by the destination node and any other nodes which are probed with the request to respond to the request appropriately.

According to a specific implementation and as mentioned above, the interconnection controller in each cluster appears to the other processors in its cluster as just another processor in the cluster. However, the portion of the shared memory space associated with the interconnection controller actually encompasses the remainder of the globally shared memory space, i.e., the memory associated with all other clusters in the system. That is, from the perspective of the local processors in a particular cluster, the memory space associated with all of the other multi-processor clusters in the system are represented by the interconnection controller(s) in their own cluster.

According to an even more specific implementation which will be described with reference to Fig. 5, each cluster has five nodes (e.g., as shown in Fig. 2) which include four processors 202a-d and an interconnection controller 230, each of which is represented by a 3-bit node ID which is unique within the cluster. As mentioned above, each processor (i.e., cluster node) may represent a number of sub-units including, for example, CPUs, memory controllers, etc.

An illustration of an exemplary address mapping scheme assuming such a cluster configuration is shown in Fig. 5. In the illustrated example, it is also assumed that the global memory space is shared by 4 such clusters also referred to herein as quads (in that each contains four local processors). As will be understood, the number of clusters and nodes within each cluster may vary according to different implementations.

To extend the address mapping function beyond a single cluster, each cluster maps its local memory space, i.e., the portion of the global memory space associated with the processors in that cluster, into a contiguous region while the remaining portion of the global memory space above and below this region is mapped to the local interconnection controller(s). The interconnection controller in each cluster maintains two mapping tables: a global map and local map. The global map maps outgoing requests to remote clusters. The

local map maps incoming requests from remote clusters to a particular node within the local cluster.

Referring now to Fig. 5, each local cluster has a local memory map (501-504), which maps the local memory space (i.e., the contiguous portion of the global memory space associated with the local processors) into the respective nodes and maps all remote memory spaces (i.e., the remainder of the global memory space) into one or two map entries associated with the local interconnection controller(s), e.g., Node 4 of Quad 3. Each node in the local cluster has a copy of the local map. The interconnection controller in each cluster also maintains a global map (505-508) relating these remote memory spaces with each of the other clusters in the system. Each interconnection controller uses its copy of the local map (509-511) to map requests received from remote clusters to the individual nodes in its cluster.

An exemplary transaction described with reference to Fig. 5 may be illustrative. In this example, Node 2 in Quad 3 generates a request that maps (via map 501) to the local interconnection controller (i.e., Node 4). When the interconnection controller receives this request, its global map 505 maps the address to Quad 2. The interconnection controller then forwards the request to Quad 2. The interconnection controller at Quad 2 uses its local memory map to determine the proper node to target for the request – Node 1 in this example.

In a particular implementation, each processor or cluster node is limited to eight memory map registers. The scheme described above with reference to Fig. 5 requires four entries for the local memory space and at most two registers for remote space. Therefore, according to more specific implementations, the two remaining entries can be used to subdivide regions. The eight mapping register limit requires that all memory local to a quad be allocated within a contiguous block. The interconnection controller's local memory map in such implementations is also eight entries. However, the size of the interconnection

controller's global map size is determined by the number of clusters in the system.

According to various implementations, the memory mapped I/O space is mapped by an identical set of mapping registers.

As described above, on the local cluster level, information from address mapping
5 tables is used to identify the appropriate link on which to transmit information to a destination node within the cluster. To effect transmissions between clusters using the global mapping described above, a similar mechanism is needed. Therefore, according to various implementations, in addition to the local routing tables associated with each node in a cluster, the interconnection controller maintains global routing information which maps the
10 other clusters in the system to the various point-to-point transmission links interconnecting the clusters (e.g., links 111 of Fig. 1A).

According to a specific implementation, two types of local routing tables are employed: one for directed packets and one for broadcast packets. Each table (e.g., tables 406 of Fig. 4) maintains a mapping between target nodes and links. For directed packets, a
15 separate table is used for request and for responses. This allows responses to be routed back to the requester along the same path as the request. Maintaining the same route simplifies debugging and is not required for correctness. For broadcast packets, the corresponding table indicates on which links the broadcast packet is forwarded. A broadcast packet may thus be routed to multiple links.

20 In a particular implementation of the interconnection controller, its local tables map a local destination node to one of four links for directed packets and any number of links for broadcast packets. The interconnection controller also maintains a global routing table which maps remote destination clusters to a particular remote link. According to a particular implementation, the interconnection controller also supports multicast of packets at the
25 global routing level.

A specific implementation of a routing mechanism will now be described with reference to Figs. 6A and 6B. System 600 of Fig. 6A includes four clusters each having a plurality of local nodes including nodes N_0 and N_1 . The table of Fig. 6B combines all of the local and global routing tables of the system for illustrative purposes.

5 As part of an exemplary transaction, a CPU 602 at node N_0 in Cluster 0 generates a packet directed to a CPU 604 at node N_0 in the Cluster 3. This packet could be, for example, a memory request that maps to a memory controller at that node. Because CPU 602 has no knowledge of anything outside of its cluster, it generates the packet targeting node N_1 in Cluster 0 (i.e., the local interconnection controller 606) as the destination. As discussed
10 above, this is due to the fact that the local memory map owned by node N_0 (see the relevant portion of the table of Fig. 6B) identifies node N_1 as corresponding to all memory owned by remote clusters. Interconnection controller 606 receives the packet, uses its global address map (e.g., as described above) to determine that the final destination of the packet is Cluster 3, and generates a remote packet targeting Cluster 3. Then, using its global routing table
15 (i.e., relevant portion of Fig. 6B), interconnection controller 606 determines that this packet must be sent out on link L_1 . Similar to the local routing mechanism described above, information identifying the source and destination cluster is appended to the packet.

When interconnection controller 608 at Cluster 1 receives the packet, it also determines that the packet is destined for Cluster 3 and determines from its global routing
20 table (Fig. 6B) that link L_2 must be used to send the packet. Interconnection controller 610 at Cluster 3 receives the packet, determines that the packet is targeting the local cluster, and uses its local routing table (Fig. 6B) to determine that local link L_0 must be used to send the packet to its destination. CPU 604 at node N_0 then receives the packet via link L_0 .

According to specific implementations in which the node ID space is a 3-bit ID space, this

multi-level routing mechanism can be extended to eight local nodes with no specific limit on the number of clusters.

Some implementations also address the issue of transaction identification in a system having a plurality of multi-processor clusters. In general, the importance of the unique
5 identification of transactions in a multi-processor environment is understood. And where the transaction identification or tag space is limited, mechanisms to extend it are needed to enable the interconnection of more than the maximum number of processors supported by the limited tag space. That is, in an environment with a plurality of clusters operating with identical local transaction tag spaces, there is a potential for more than one transaction to be
10 generated in different clusters simultaneously with the identical tag. Where those transactions occur between nodes in different clusters, the potential for conflict is obvious. Therefore, some implementations provide mechanisms which extend the local tag spaces such that each transaction in the multi-cluster system is uniquely identified.

More specifically, these implementations map transactions from the local transaction
15 tag space to a larger global transaction tag space. As described above, the local tag space is specified using the node ID, the unit ID, and a transaction ID. On top of that, the global tag space is specified using a global cluster ID and a global transaction ID. According to one implementation, the interconnection controllers in the system use their pending buffers to simplify the allocation and management of the mapping and remapping actions. According
20 to an even more specific implementation and as will be described, additional protocol management is used to maintain the uniqueness of the global transaction tags.

According to a specific implementation, all transactions within a cluster are tagged with a unique ID generated by the requesting node. The processors in each cluster which are not the interconnection controller support a 3-bit node ID, a 2-bit unit ID and a 5-bit
25 transaction ID. The combination of these fields creates a 10 bit tag which is unique within

the cluster. The unit ID represents sub-units within a node. It should be noted that a particular node may or may not include a processor as one of its sub-units, e.g., the node might contain only memory.

According to one implementation, to extend to the transaction tag space beyond the local cluster, each cluster's interconnection controller maps each its cluster's local tag space into the global tag space using a Q-bit Cluster ID and a T-bit Transaction ID. In the exemplary system in which each cluster has a 5-bit transaction ID and there are four clusters, T might be 7 and Q might be 2.

According to one implementation illustrated in Fig. 7, the local to global mapping process is accomplished as follows. New outgoing transactions, i.e., locally generated transactions involving remote clusters, are received by the cluster's interconnection controller (702) and are allocated an entry in the protocol engine's global pending buffer (704), e.g., pending buffer 309 of Fig. 3. The buffer entry identifier is unique for each such transaction, and therefore is used as a unique global transaction tag. According to a specific implementation, the number of entries in the pending buffer cannot exceed the size of the global transaction tag space for this mechanism. Also included in the pending buffer entry is the local transaction tag. Thus, the information mapping the transaction from the local tag space to the global tag space is implicitly contained in the pending buffer. The outgoing transaction is then tagged with the global transaction tag, along with a cluster ID that is specified during initial system configuration (706). When incoming transmissions associated with the transaction are received (708), the global transaction tag is used to index into the global pending buffer (710). If a local transaction tag is required (712), the local tag from the indexed pending buffer entry is used to tag the transaction (714).

According to a specific implementation illustrated in Fig. 8, the global to local mapping process is as follows. New incoming transactions, i.e., remotely generated

transactions involving the local cluster, are received by the local cluster's interconnection controller (802) and assigned a local transaction tag using a similar mechanism as described above (804). In this case, a pending buffer entry is allocated in the protocol engine's local pending buffer (806), and an entry containing the global tag is inserted into the newly
 5 allocated pending buffer entry (808). When outgoing transmissions associated with the transaction are received (810), the local transaction tag is used to index into the local pending buffer to obtain the proper global tag (812) which is used if subsequent outgoing transmissions are required (814).

An example of the tag mapping mechanisms will now be described with reference to
 10 Fig. 9. Fig. 9 is a diagrammatic representation depicting the transactions for a data request from a local processor sent to a non-local cluster using an interconnection controller. It will be understood that the transactions depicted are merely exemplary and should not be used to limit the scope of the invention. The multi-cluster system includes a request cluster 900, a home cluster 920, and a remote cluster 940. The home cluster 920 and the remote cluster
 15 940 are referred to herein as non-local clusters. Processors and interconnection controllers associated with local and non-local clusters are similarly referred to herein as local processors, local interconnection controllers, non-local processors, and non-local interconnection controllers, respectively.

According to various implementations, processor 901-1 in a local cluster 900 sends a
 20 data access request such as a read request to an interconnection controller 903-1. The requesting node 901-1 generates the request with a local tag (e.g., tag 902) in the cluster's local tag space. The interconnection controller 903-1 maps the outgoing request into the global tag space (e.g., using global tag 904) and the mapping is saved in the pending buffer (e.g., buffer 309 of Fig. 3) of interconnection controller 903-1 as described above (e.g., table
 25 insertion: local tag 902/global tag 904). Interconnection controller 903-1 then forwards the

request to an interconnection controller 921-1 in a home cluster 920. Incoming responses associated with the transaction (e.g., 908 and 910) are mapped from the global tag space back to the local tag space using a table lookup in the interconnection controller 903's pending buffer.

5 At home cluster 920, incoming requests (e.g., 912) are mapped by interconnection controller 921-1 into the local tag space (e.g., using tag 914) and the mapping is saved in the pending buffer of controller 921-1 (e.g., table insertion: global tag 904/local tag 914). Outgoing responses from home cluster 920 (e.g., 908 and 910) are mapped from the local tag space back to the global tag space using a table lookup in interconnection controller 921's
10 pending buffer. Incoming responses to the home cluster are mapped from the global tag space back to the local tag space using a table lookup in interconnection controller 921's pending buffer.

 In this exemplary transaction, interconnection controller 921-1 forwards the access request to a memory controller 923-1 also associated with home cluster 920. At this point,
15 memory controller 923-1 locks the memory line associated with the request. In one example, the memory line is a unique address in the memory space shared by the multiple processors in request cluster 900, home cluster 920, and remote cluster 940. Memory controller 923-1 generates a probe associated with the data access request and forwards the probe to local nodes associated with cache blocks 925 and 927 as well as to interconnection
20 controller 921-2.

 Interconnection controller 941-1 at remote cluster 940 maps global tags (e.g., tag 922) into its local tag space using local tags (e.g., tag 924). Responses (e.g., 926) are mapped from the local tag space back to the global tag space using a table lookup in interconnection controller 941's pending buffer. In response to the probe from home cluster
25 920, interconnection controller 941-1 probes local nodes associated with cache blocks 945,

947, and 949. Similarly, interconnection controller 903-2 associated with request cluster 900 receives a probe and forwards the probe to local nodes associated with cache blocks 905, 907, and 909. Processor 901-2 receives probe responses from the local nodes associated with cache blocks 905, 907, and 909.

5 According to various implementations, interconnection controller 921-3 accumulates probe responses and sends the probe responses to interconnection controller 903-3, which in turn forwards the probe responses to the processor 901-3. Interconnection controller 921-4 also sends a memory controller read response 910 to interconnection controller 903-4, which forwards the read response to processor 901-4. After receiving all responses and the fetched
10 data, processor 901-4 sends a source done response to interconnection controller 903-5. Interconnection controller 903-5 forwards the source done message to interconnection controller 921-5. Interconnection controller 921-5 in turn sends a source done message to memory controller 923-2.

 According to a specific implementation, interconnection controller 921-5 also
15 acknowledges the Source Done transaction (918) with a tag release transaction (928) to controller 903-6 at the requesting cluster that allows it to reuse the tag. As described below, this acknowledgment prevents the requesting cluster from reusing a global tag until the global tag is no longer in use in the system.

 It should be noted that the above-described tag mapping mechanism works for all
20 transaction types with the exception of the Source Done transaction. In this case, the outgoing SD transaction received by the interconnection controller at the requesting cluster is tagged with the local node's identifier. As a result, the interconnection controller at the requesting cluster must search its pending buffer for an entry with a matching local tag and then use the index as the global tag. The same mechanism is required when the Source Done

transaction reaches the home cluster. In this case, the pending buffer at the home quad is searched for a matching global tag and the index is used as the local tag.

According to a specific implementation, the final tag release transaction (e.g., 928) is provided to inform the requesting cluster when a given tag may be reused. This tag release mechanism is for preventing ambiguity of Source Done (SD) transactions at the home cluster. For example, without such a mechanism, after the requesting cluster completes a transaction and a SD transaction is sent to the home cluster (e.g., 918), a new transaction can be generated at the requesting cluster with the same global tag. A transmission associated with this transaction could potentially be received at the home cluster and inserted into the home cluster pending buffer prior to the receipt of the first SD transaction as they travel on different virtual channels. Under such circumstances, the pending buffer at the home cluster would have multiple entries with the same global tag. Requiring the tag release transaction forecloses this possibility.

As described above, the interconnection controller described herein facilitates the merger of independent cache-coherent clusters into a larger cache coherent system.

According to a particular implementation, these interconnection controllers are embodied as ASIC chips and the following description assumes this. However, it should be understood that neither the interconnection engines nor their included functionalities are so limited.

According to various implementations, protocol engines are blocks of hardware on the interconnection controller ASIC chip. The functionality of these engines are governed by associated microcode and relate generally to managing transaction flows. Generally speaking, a protocol engine looks at each packet received by the interconnection controller and makes decisions regarding the appropriate handling of the packet and any actions/response which need to be taken. Having a single protocol engine to manage transactions (e.g., as shown in Fig. 3) can be a transaction processing bottleneck.

Therefore, as mentioned above and according to a specific implementation, each interconnection controller comprises multiple protocol engines to improve the transaction processing throughput of the system. According to the described implementation, each of the protocol engines comprises substantially identical hardware blocks, each being
5 programmed by the associated microcode to perform a particular type of transaction processing. More specifically, in the described implementation there are three different types of protocol engines in each interconnection controller which will be referred to herein as remote memory protocol engines, local memory protocol engines, and special protocol engines, respectively. Fig. 10 is a simplified block diagram of an exemplary interconnection
10 controller 1000 designed according to such an implementation. It will be understood that interconnection controller may be used in the processor cluster of Fig. 2 in place of interconnection controller 230.

Remote memory protocol engines (RMPEs) 1002 are responsible for processing transactions which target remote memory, i.e., memory associated with another cluster, and
15 all subsequent transactions in that particular transaction flow. Local memory protocol engines (LMPEs) 1004 are responsible for processing transactions which target local memory, i.e., memory associated with the local cluster with which the interconnection controller is associated, and all subsequent transactions in the flow. Special protocol engine 1006 is responsible for processing interrupts, I/O accesses, configuration accesses, etc.
20 According to the implementation shown in Fig. 10, each interconnection controller includes 2 remote engines, 4 local engines, and one special engine. It will be understood however, that these numbers are merely exemplary, and the optimal number of each engine type may vary according to the manner in which transactions are distributed across the engines.

Interconnection controller has a coherent protocol interface 1007 that allows the
25 interconnection controller to communicate with other processors in the cluster as well as

external processor clusters. Pending buffer 1009 and noncoherent interface 1011 may have functionalities similar to those for similar processor blocks described above with reference to Fig. 3. According to various implementations, interface 1007 is implemented either as a full crossbar or as separate receive and transmit units using components such as multiplexers and buffers. According to specific implementations, interface 1007 may be configured to effect the routing of transaction packets to particular protocol engines according to the implementations described below.

Distributing the transaction processing work load among the various protocol engines in an interconnection controller may be done in a variety of ways. According to a specific implementation, the memory mapping tables are employed to assign each protocol engine to a particular address range or set of ranges within the global memory space, i.e., the memory space shared by all of the processor clusters. That is, the initial request or probe packets in a transaction flow include address information which is used to map the transaction to a particular protocol engine according to the address range(s) associated with each engine. In one implementation, the assignment of the address ranges to the various protocol engines associated with the plurality of clusters occurs when the system is first initialized. According to other implementations, assignment or reassignment may occur after system initialization.

According to one implementation, multiple node IDs are used for the interconnection controller in each cluster. This is enabled in a particular implementation by the fact that the capacity of the node ID space associated with each cluster exceeds the number of nodes in a cluster. For example, an implementation described above with reference to Fig. 5 assumed five nodes in each cluster including one associated with the interconnection controller. However, if multiple nodes are associated with the interconnection controller, each node can be associated with one of the remote engines, the memory mapping tables associated with

the local nodes within the cluster can be used to map the memory regions for the nodes associated with the interconnection controller to the remote engines. Similarly, these mapping tables may be used to map the memory regions associated with the other nodes in the cluster, i.e., the local nodes, to the local engines.

5 According to a specific implementation, the individual protocol engines in the interconnection controller are referenced by node and unit ID pairs. Thus, seven protocol engines are referenced using only four node IDs. According to this implementation, RMPEs 1002 are referenced with two node IDs and one unit ID, LMPEs 1004 are referenced with the same two node IDs and two unit IDs, and special protocol engine 1006 is referenced
10 using a single node ID. For example, RMPE0 is represented by node and unit ID combination {4, 2}, RMPE1 by {5, 2}, LMPE0 by {4, 0}, LMPE1 by {4, 1}, LMPE2 by {5, 0}, LMPE3 by {5, 1}, and the special engine by {6, x}. In this implementation, the unit IDs 0, 1, 2, and 3 in the other local nodes correspond to a CPU, a CPU, a memory controller, and an I/O bridge, respectively.

15 As described above, in a particular cluster the contiguous portion of the global memory space associated with remote clusters, i.e., the remote memory space, is associated with or mapped to the node or set of nodes corresponding to the local interconnection controller. The RMPEs in each interconnection controller are mapped to the same portion of the global memory space. More specifically and according to one implementation, a set of
20 mapping tables is maintained which maps these engines to the remote memory space. According to various implementations, any of a variety of mapping schemes may be employed to distribute the transaction processing work load among the engines. According to specific implementations, alternating and adjacent portions of the remote memory space are alternatively assigned to the multiple RMPEs.

In addition, a set of mapping tables is maintained which maps the nodes and units associated with the local cluster to the multiple LMPEs of the local interconnection controller. As with the mapping of the RMPEs, the mapping of the LMPEs may be accomplished in a variety of ways. An example of such a mapping is shown in Fig. 11.

5 In this example, the protocol engines in Cluster 2, i.e., the local cluster, are mapped to a physical memory space 1100 associated with a four cluster system. Each cluster is assumed to include four local nodes and an interconnection controller which corresponds to two additional local nodes. As described above, each of the four local nodes may comprise multiple units including, for example, a memory controller and a host bridge. Each of the
10 clusters corresponds to a contiguous portion of memory space 1100. Two DRAM mapping tables associated with Cluster 2 map the portions of memory space 1100 corresponding to Clusters 0 and 1 to the remote engines of Cluster 2, i.e., RMPE1 and RMPE0. Another two DRAM mapping tables map the portion of memory space 1100 corresponding to Cluster 3 to RMPE0 and RMPE1 as well. Another four DRAM mapping tables map the portions of
15 memory space corresponding to Cluster 2 to the four local nodes (other than the nodes associated with the interconnection controller) in Cluster 2.

According to various implementations, these mappings may follow any of a variety of schemes. For example, as mentioned above, alternating and adjacent portions of the memory space associated with a particular cluster may be assigned to RMPE0 and RMPE1.
20 These alternating portions may be a memory line, a memory page, multiple pages, etc., with the mappings striding on the boundaries, e.g., the line or page boundaries.

Certain types of transaction packets, e.g., packets associated with Source Done transaction 918 of Fig. 9, are not amenable to a straightforward address range assignment scheme because they do not include associated address information. As described above,
25 this may be dealt with by broadcasting such packets to all protocol engines in the system

which might be responsible for the particular transaction. However, this may be too inefficient and, in fact, partially defeats the purpose of dividing up the work load among multiple engines.

Therefore, according to a specific implementation, the destination node and destination unit fields in transaction packets which target remote memory spaces are set by the responsible remote protocol engine to ensure the packets are properly mapped to the correct local protocol engine in the remote cluster to which they are directed. Similarly, the destination node and destination unit fields in transaction packets which target local memory spaces are set by the responsible local protocol engine to ensure the packets are properly mapped to the correct remote protocol engine in the remote cluster to which they are directed. Alternatively, these fields may be set only in packets which do not have an address specified, e.g., Source Done packets.

According to a specific implementation, the responsible protocol engine at the cluster from which the packets originate maps the destination node and destination unit values generated by the local nodes to destination node and destination unit values that correspond to a node and a unit (and thus a protocol engine) in the cluster to which the packets are directed. For packets having address values specified, this mapping may be accomplished with reference to the global address to node mapping information available to the interconnection controller. For packets having no address values specified, this mapping may be accomplished with reference to entries in the interconnection controller's pending buffer corresponding to the node, unit, and transaction ID specified in the packets.

According to a specific implementation, an interleaved mapping from a protocol engine in one cluster, e.g., a request cluster, to a protocol engine in a cluster to which a packet is directed, e.g., a home cluster, is accomplished with reference to specific bits in the target address. That is, for example, a request packet in the request cluster targeting remote

memory is mapped to either RMPE0 or RMPE1 depending upon the state of a particular bit in the address to which the packet is directed. The remote engine to which the packet is mapped remaps the packet's destination node and unit IDs as discussed above. When received by the interconnection controller in the home cluster, the packet is mapped to one of the four local protocol engines in accordance with the destination node and unit IDs.

An exemplary mapping might be represented in terms of address values as follows:

$$\text{RMPE0} \Rightarrow \text{Addr}[13] = 0 \quad \Leftrightarrow \quad \text{LMPE0} \Rightarrow \text{Addr}[13:12] = 00$$

$$\Leftrightarrow \quad \text{LMPE2} \Rightarrow \text{Addr}[13:12] = 10$$

$$\text{RMPE1} \Rightarrow \text{Addr}[13] = 1 \quad \Leftrightarrow \quad \text{LMPE1} \Rightarrow \text{Addr}[13:12] = 01$$

$$\Leftrightarrow \quad \text{LMPE3} \Rightarrow \text{Addr}[13:12] = 11$$

This representation indicates that if address bit 12 of the address corresponding to the destination node identified in the packet is a "0," the transaction will be managed by RMPE0 in the request cluster and by either LMPE0 or LMPE2 in the home cluster (depending upon the state of address bit 13). Similarly, if address bit 12 is a "1," the transaction will be managed by RMPE1 and either LMPE1 or LMPE3 (depending upon the state of address bit 13). Bit 12 corresponds to the page boundary in a 40-bit address. According to another implementation using a 40-bit address, bits 6 and 7 are used to determine the interleaving with bit 6 corresponding to the line boundary.

A specific implementation will now be described with reference to the flowchart of Fig. 12. As described above, an implementation will be assumed in which packets generated

in a local cluster are mapped to a protocol engine according to the target address of the packets. This target address is either derived directly from the packets themselves or by reference to a pending buffer entry (e.g., for packets specifying no address).

For directed packets, i.e., packets specifying a particular node/unit ID pair, the mapping to a particular protocol engine is automatically done by the local node CPU generating the packet according to its mapping tables. That is, these packets are automatically sent to the correct protocol engine by virtue of the node and unit IDs generated by the requesting node. For broadcast packets, i.e., packets not specifying a particular node/unit ID pair, the mapping to one of the protocol engines is done by the interconnection controller based on the address information in the packet. As mentioned above, whether the mapping is accomplished at the local node based on the node/unit ID pair specified by the local CPU, or at the interconnection controller based on the address information in the packet, many different mapping schemes may be employed.

Packets received from a remote cluster are mapped to a protocol engine in the receiving cluster according to the local node and unit to which the packets are directed. As indicated above, the node and unit values are set by the protocol engine in the cluster from which the packets originated, i.e., the requesting cluster. Notwithstanding the foregoing, it should be understood that variations in the mapping mechanism may be employed. For example, packets received from a remote cluster may be mapped to protocol engines in the receiving cluster based on the target address rather than the destination node and unit. Therefore, the following description should be understood to be exemplary.

Fig. 12 is a flowchart illustrating an exemplary implementation by which the processing of transactions in a computer system having multiple multi-processor clusters is distributed over multiple protocol engines in each cluster. For exemplary purposes, various of the details of the system described above with reference to Figs. 1-11 will be assumed.

However, it will be understood that the illustrated technique may be employed in a wide variety of multi-processor systems.

When a transaction packet (e.g., a memory request or a probe) is generated (1202) by a local node in a particular processor cluster, the packet is routed to the appropriate local node(s) and unit(s) using the address and link mapping techniques described above. Where the packet targets a memory address associated with a remote cluster, the packet is directed to one of the remote protocol engines associated with the local interconnection controller. Where the packet targets a memory address associated with the local cluster, the packet is directed (e.g., in the case of a broadcast packet like a probe) to one of the local protocol engines associated with the local interconnection controller. If the packet is a directed packet (1203), i.e., a packet identifying a specific node and unit (e.g., a memory request), the packet (and subsequent packets corresponding to the associated transaction) is transmitted to the identified one of the plurality of local nodes and units associated with interconnection controller, each of which corresponds to protocol engine (1204). This may be accomplished using the local memory map associated with each local node. As mentioned above, the interleaving of the addresses across the protocol engines may be achieved in a variety of ways. Also as mentioned above, for packets not specifying an address (e.g., Source Done packets), the address may be determined with reference to any pending buffer entries corresponding to the transaction ID field identified in the packet.

If, on the other hand, the packet is a broadcast packet (1203), i.e., a packet which does not specify a particular node or unit ID (e.g., a probe), the local interconnection controller receives the packet and maps it (and any subsequent packets in the transaction flow) to one of the protocol engines according to a mapping table which maps address ranges to the protocol engines according to any of a variety of mapping schemes. As discussed above, if the target address corresponds to local memory, the packet is mapped to

the corresponding one of the local protocol engines. If, on the other hand, the target address corresponds to remote memory, the packet is mapped to the corresponding one of the remote protocol engines.

Regardless of how the mapping is effected, the protocol engine to which the packet is assigned determines how to process the packet (1206), maps the destination node and unit ID fields in the packet to the node and unit IDs in the remote cluster to which the address corresponds (1208), and transmits the packet to the remote cluster (1210). The interconnection controller in the remote cluster receives the packet (1212) and, according to the destination node and unit ID fields set by the interconnection controller in the request cluster, maps the processing of the packet to one of its protocol engines (1214). According to a specific implementation, each node/unit ID pair corresponds to one of the protocol engines in the remote cluster. The protocol engine in the remote cluster then determines how to process the packet (1216), and transmits the packet to the appropriate local node(s) (1218).

As should be apparent by reference to the preceding discussion, the flowchart of Fig. 12 not only describes the mapping of a transaction to a remote protocol engine in a local cluster and a local protocol engine in a remote cluster, but also the mapping of a transaction to a local protocol engine in a local cluster and a remote protocol engine in a remote cluster. That is, for example, Fig. 12 describes the process in which a local node in a local cluster generates a memory request targeting memory in a remote cluster, and by which that memory request is mapped to one of the remote protocol engines in the local interconnection controller, mapped by the local interconnection controller to the appropriate node/unit ID in the remote cluster, and mapped by the interconnection controller in the remote cluster to the corresponding local protocol engine.

Fig. 12 also describes, for example, the process in which a local node in a local cluster generates a probe targeting memory in the local cluster, and by which that probe is mapped to one of the local protocol engines in the local interconnection controller, mapped by the local interconnection controller to the appropriate node/unit ID(s) in at least one remote cluster, and mapped by the interconnection controller(s) in the remote cluster(s) to the corresponding local protocol engine(s).

As discussed above, the interconnection controller in each cluster tracks transactions using its pending buffer to achieve a variety of functionalities. According to various embodiments of the invention, this transaction tracking capability is further leveraged to facilitate the generation of transaction latency data. According to a specific embodiment, each interconnection controller in the system includes at least one set of counters, at least one of which measures the latencies for specific transactions (referred to herein as latency counters), and a plurality of which organize the latencies into “buckets” (referred to herein as histogram counters). In this way, a distribution of latency data may be generated. A block diagram of such an interconnection controller 1300 is shown in Fig. 13.

According to a more specific embodiment, the histogram counters are an array of N counters 1302 designated sequentially from 0 to $N-1$, each representing a particular latency range within a window. The window and ranges over which the latency data are to be obtained are described using programmable parameters referred to herein as ***Bucket Start*** and ***Bucket Size***. In addition, two signals referred to herein as ***Transaction Start*** and ***Transaction End*** are hardware generated signals that indicate the beginning and end of a transaction, respectively. Of the N counters, counter 0 is referred to as the underflow counter and counter $N-1$ is referred to as the overflow counter.

In addition to the array of histogram counters, there is at least one latency counter 1304 which is used to count clock cycles between the assertion of the ***Transaction Start*** and

Transaction End signals. In theory, there can be as many latency counters as there can be ongoing transactions in the system that are visible to the hardware, with each latency counter being dedicated to a particular transaction. According to such an embodiment, the **Transaction Start** and **Transaction End** signals would be unique to each latency counter.

5 For the remainder of this description it is assumed that only a single latency counter is being used.

According to a specific embodiment, a transaction begins (and the **Transaction Start** signal is asserted) when the interconnection controller receives a probe and places the corresponding entry in its pending buffer. A transaction ends (and the **Transaction End** signal is asserted) when the interconnection controller receives the accumulated responses to the forwarded probe and removes the pending buffer entry. It will be understood that this embodiment of the invention ignores the portions of each transaction preceding receipt of the probe by the interconnection controller and following deletion of the pending buffer entry. However, the duration of these portions of each transaction is very small as compared to the time required to forward the probe to other clusters and accumulate the probe responses, and thus may be ignored without unduly affecting the accuracy of the latency measurement.

It should also be noted that the types of transactions for which latency data may be generated according to various embodiments of the invention may include both memory transactions as well as transactions involving I/O devices.

20 Referring to flowchart 1400 of Fig. 14, when the **Transaction Start** signal is asserted (1402), the latency counter is reset (1404) and begins counting clock cycles (1406) until the **Transaction End** signal indicates the end of the transaction (1408). The resulting latency counter value is then placed in corresponding bucket associated with the histogram counters (1410). That is, the histogram counter having the latency range which includes the latency counter value is incremented. The process is then repeated for another transaction until the

25

run is complete (1412). In this way, a count is accumulated for each latency range within the overall window. The counts may then be accessed at virtually any time by simply reading configuration registers associated with the counters (1414).

According to a specific embodiment, *Transaction Start* and *Transaction End* are programmable in that they can be made to correspond to any arbitrary event associated with a transaction for which a latency measurement is desired. For example, *Transaction End* can be made to correspond to particular types of memory request responses from remote clusters, e.g., a “dirty” or a “shared” response. In another example, *Transaction End* can be made to correspond to responses from particular I/O devices. In this way, detailed latency data may be generated for particular types or even sets of transactions.

Bucket Start can either be set to 0 or can be set to have an offset value. *Bucket Size* corresponds to the width of the latency range associated with each bucket. These parameters may be manipulated to adjust the “granularity” with which the data are generated to suit a particular application. For example, based on experienced guesses, the system evaluator might initially set *Bucket Start* to 0 and *Bucket Size* to 100 clock cycles. So, for $N = 8$, bucket $N-1$ would correspond to a latency range of 700 to 800 clock cycles. If the results of an initial run using these values indicates that most of the measured latencies lie in the range of 400 to 600 clock cycles, the system evaluator can then “drill down” by, for example, setting *Bucket Start* to 400 and *Bucket Size* to 25, to generate more detailed latency data for this window. This could be done iteratively to get a very clear picture of where the latencies lie.

According to a specific embodiment, the range of each histogram counter in the counter array is represented as follows:

	Counter	Description	Range
	Counter N-1	Overflow	$\text{Bucket Start} + (N-2) * \text{Bucket Size} \leq \text{Count} < \infty$
	Counter N-2		$\text{Bucket Start} + (N-3) * \text{Bucket Size} \leq \text{Count} < \text{Bucket Start} + (N-2) * \text{Bucket Size}$
5	Counter N-3		$\text{Bucket Start} + (N-4) * \text{Bucket Size} \leq \text{Count} < \text{Bucket Start} + (N-3) * \text{Bucket Size}$
	:		
	Counter 4		$\text{Bucket Start} + (3 * \text{Bucket Size}) \leq \text{Count} < \text{Bucket Start} + (4 * \text{Bucket Size})$
	Counter 3		$\text{Bucket Start} + (2 * \text{Bucket Size}) \leq \text{Count} < \text{Bucket Start} + (3 * \text{Bucket Size})$
	Counter 2		$\text{Bucket Start} + (1 * \text{Bucket Size}) \leq \text{Count} < \text{Bucket Start} + (2 * \text{Bucket Size})$
10	Counter 1		$\text{Bucket Start} \leq \text{Count} < \text{Bucket Start} + (1 * \text{Bucket Size})$
	Counter 0	Underflow	$0 \leq \text{Count} < \text{Bucket Start}$

15 The pseudo-code of counter operation according to a specific embodiment of the invention is as follows:

```

20 IF (Transaction Start==TRUE) && (Latency Counter available==TRUE) THEN
    RESET Latency Counter
IF (Transaction END==TRUE) THEN
    IF (Latency Counter < Bucket Start)
        INCREMENT Underflow counter
    ELSE IF (Latency Counter >= Bucket Start + (N-2)*Bucket Size) THEN
25         INCREMENT Overflow counter
    ELSE
        INCREMENT Counter Corresponding to Bucket value of
        Latency Counter
ELSE
30     INCREMENT Latency Counter

```

Specific embodiments of the invention may be characterized by various advantages over conventional approaches to measuring latency. One such advantage relates to the fact that particularly meaningful information regarding system latency may be obtained beyond mere minimums, maximums, and averages. That is, the captured latency data may be represented as a distribution over a window, the size and precision of which may be manipulated to provide more or less detail. This allows the designer or a system administrator to acquire a highly accurate picture of latency distribution when evaluating system performance, and to focus more or less accurately within particular windows and ranges depending upon the information desired.

Another advantage associated with some embodiments relates to the fact that latency data may be generated without user intervention in a manner which minimizes or eliminates overhead penalties or system perturbations associated with conventional methods. Once initially configured, the histogram counters associated with such embodiments require no
5 intervention, thereby producing uncorrupted results.

According to a specific embodiment implemented in a multi-processor system which employs an interconnection controller having multiple protocol engines (e.g., interconnection controller 1000 of Fig. 10), a set of latency and histogram counters is associated with each protocol engine in the interconnection controller. Because each of the
10 remote and local protocol engines is responsible for transactions involving remote and local processors, respectively, latency data may be generated separately for these transaction types using this implementation.

The latency data may be generated during the design phase, or even after the system has been delivered to the customer, to measure and/or optimize system performance.

15 According to various embodiments, the integrated nature of the techniques greatly simplifies or eliminates the instrumentation required by conventional approaches to latency measurement. Various embodiments also allow latency data to be conveniently and flexibly generated at any time.

The latency data generated by various embodiments of the invention may also be
20 used for a variety of purposes. For example, latency data may be used by an operating system on an ongoing basis to move data sets around among the various memory blocks associated with different processors to achieve more efficient operation. That is, the operating system might adjust its data placement algorithms such that a more desirable latency distribution is achieved and maintained. More generally, latency data can be used to facilitate the dynamic

tuning of virtually any parameter associated with a system's operation. Such an approach may be used, for example, to maintain a particular latency distribution over time.

While the invention has been particularly shown and described with reference to specific embodiments thereof, it will be understood by those skilled in the art that changes in the form and details of the disclosed embodiments may be made without departing from the spirit or scope of the invention. For example, specific embodiments have been described herein with reference to particular multi-processor architectures having point-to-point communication infrastructures. It will be understood, however, that the specific architectures described above are illustrative and that other point-to-point architectures (including single cluster multi-processor systems) may employ the inventive techniques described and claimed herein.

It will be further understood, that the present invention applies more generally to a wide variety of computer architectures including, for example, computer systems having shared-bus architectures. An example of such a bus-based system 1500 is shown in Fig. 15. As indicated by the dashed lines associated with some of the processors, the invention may be implemented in single or multi-processor shared-bus architectures. According to such alternative embodiments, the counters could be implemented in a variety of locations such as, for example, at the memory controllers or at the processing nodes themselves (e.g., in the host bridge).

Still more generally, the present invention may be practiced in any electronic or data processing system for which such the latency measuring capabilities enabled are useful without departing from the scope of the invention.

It should also be understood that the various embodiments of the invention may be implemented or represented in a wide variety of ways without departing from the scope of the invention. That is, for example, the interconnection controller described herein may be

represented (without limitation) in software (object code or machine code), in varying stages of compilation, as one or more netlists, in a simulation language, in a hardware description language, by a set of semiconductor processing masks, and as partially or completely realized semiconductor devices. The various alternatives for each of the foregoing as
5 understood by those of skill in the art are also within the scope of the invention. For example, the various types of computer-readable media, software languages (e.g., Verilog, VHDL), simulatable representations (e.g., SPICE netlist), semiconductor processes (e.g., CMOS), and device types (e.g., ASICs) suitable for designing and manufacturing the processes and circuits described herein are within the scope of the invention.

10 Finally, although various advantages, aspects, and objects of the present invention have been discussed herein with reference to various embodiments, it will be understood that the scope of the invention should not be limited by reference to such advantages, aspects, and objects. Rather, the scope of the invention should be determined with reference to the appended claims.